Red Hat Enterprise Linux 4

Developer Tools Guide

Red Hat Enterprise Linux 4: Developer Tools Guide

Red Hat, Inc.

1801 Varsity Drive
Raleigh NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park NC 27709 USA

dev(EN)-4-HTML-RHI (2005-02-01)
Copyright © 2005 by Red Hat, Inc. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, V1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub/).
Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.
Distribution of the work or derivative of the work in any standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from the copyright holder.
Red Hat and the Red Hat "Shadow Man" logo are registered trademarks of Red Hat, Inc. in the United States and other countries.
All other trademarks referenced herein are the property of their respective owners.
The GPG fingerprint of the security@redhat.com key is:
CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

Table of Contents

1. Overview of Red Hat Enterprise Linux Developer Tools 1
2. Basic Principles of the Tools 3
2.1. Compiler and Development Tools 3
2.2. Libraries 3
2.3. Auxiliary Development Tools 4
2.4. `gcc`, the GNU Compiler Collection 4
2.5. `cpp`, the GNU Preprocessor 6
2.6. `as`, the GNU Assembler 6
2.6.1. Object Files 6
2.6.2. Assembler Directives 7
2.7. ld, the GNU Linker 7
2.8. `gcov`, the Test Coverage Tool 8
2.9. gdb, the Debugging Tool 8
2.10. make, the GNU Recompiling Tool 8
2.10.1. Example `Makefile` 9
2.11. binutils, the GNU Binary Utilities 10
2.12. Using info, the Documentation Tools 10
2.12.1. Reading info Documentation 11
2.13. Learn More About the Tools 11
3. Developing with Red Hat Enterprise Linux Developer Tools 13
3.1. Create Source Code 13
3.2. Compile, Assemble, and Link from Source Code 13
3.3. Run the Executable 13
3.4. Debug the Executable 14
3.5. Assembler Listings 16
3.6. Disassembling 16
Index 19

Chapter 1.

Overview of Red Hat Enterprise Linux Developer Tools

This manual introduces you to the Red Hat Enterprise Linux Developer Tools and its documentation set. After you read this, refer to the following documentation for details about the Red Hat Enterprise Linux Developer Tools:

- Compiler Tools
  - *Using the GNU Compiler Collection (GCC)*
  - *Using cpp, the C Preprocessor*

- Debugging Tools
  - *Debugging with GDB*

- Development Tools
  - *Using ld, the Gnu Linker*

- Auxiliary Development Tools
  - *Using as, the Gnu Assembler*
  - *Using binutils, the Gnu Binary Utilities*

HTML, PDF, and RPM versions of the manuals are available on the Documentation CD and online at http://www.redhat.com/docs/.

For more information on the tools and their history, see the following websites:

- http://sources.redhat.com/
- http://www.linuxdoc.org/
- http://www.opensource.org/products.html
- http://www.gnu.org/
- http://www.fsf.org/manual/
- http://www.openbsd.org/docum.html

redhat

Chapter 2.

Basic Principles of the Tools

Red Hat Enterprise Linux Developer Tools's C and C++ compilers, macro-assembler, debugger, binary utilities, libraries, and other development tools provide productivity, flexibility, performance, and portability. This chapter begins with a summary of the tools, then describes each tool in more detail.

To use a tool, open your system's terminal shell window and enter the tool's name as a command (`gcc`, for instance, invokes the compiler. For a summary of the Red Hat Enterprise Linux Developer Tools, refer to Section 2.1 *Compiler and Development Tools*, Section 2.2 *Libraries*, and Section 2.3 *Auxiliary Development Tools*.

To learn about working with the tools, refer to Chapter 3 *Developing with Red Hat Enterprise Linux Developer Tools*.

2.1. Compiler and Development Tools

The following lists the main tools for developing projects with Red Hat Enterprise Linux Developer Tools.

Tool	Description
`cpp`	C preprocessor (refer to Section 2.5 *`cpp`, the GNU Preprocessor*; and *Using cpp, the C Preprocessor*)
`diff, diff3, sdiff`	Comparison tools for text files.
`gcc`	ISO-conforming compiler collection (refer to Section 2.4 *`gcc`, the GNU Compiler Collection* and *Using the GNU Compiler Collection (GCC)*)
`g++`	A frontend to `gcc` that defaults to interpreting files as being C++ source files even if they look like C source files
`gcov`	Coverage analyzer, for testing code for efficiency and performance, and for use as a profiling tool (refer to *Using the GNU Compiler Collection (GCC)*)
`gdb -nw`	Debugger for debugging with command line input (refer to Section 2.9 *gdb, the Debugging Tool* and *Debugging with GDB*)
`gdb`	Debugger for making applications work better (refer to *Debugging with GDB*)
`ld`	Linker (refer to Section 2.7 *ld, the GNU Linker* and *Using ld, the Gnu Linker*)
`make`	Compilation control program (refer to Section 2.10 *make, the GNU Recompiling Tool*)
`patch`	Installation tool for source fixes

2.2. Libraries

Library	Description
`libc`	ANSI C runtime library
`libm`	C math subroutine library
`libstdc++`	C++ class library

2.3. Auxiliary Development Tools

The following components are also provided for general development:

Component	Description
`as`	Assembler (refer to Section 2.6 *`as`, the GNU Assembler* and *Using as, the Gnu Assembler*)
`info`	Online documentation tools (refer to Section 2.12 *Using info, the Documentation Tools*)
`man`	`man` pages, the standard UNIX online documentation

The GNU binary utilities provide functionality beyond the main development tools (refer to *Using binutils, the Gnu Binary Utilities*).

Utility	Description
`addr2line`	Converts addresses into file names and line numbers
`ar`	Creates, modifies and extracts from object code archives
`c++filt`	Demangles and deciphers encoded C++ symbol names
`dlltool`	Creates files for builds, using dynamic link libraries (DLLs)
`nm`	Lists symbols from object files
`objcopy`	Copies and translates object files
`objdump`	Displays information from object files
`ranlib`	Generates index to archive contents
`readelf`	Displays information about ELF format object files
`size`	Lists file section sizes and total sizes
`strings`	Lists printable strings from files
`strip`	Discards symbols

2.4. `gcc`, the GNU Compiler Collection

The GNU Compiler Collection (GCC), is a complete set of tools for compiling programs written in C, C++, Objective C, or languages for which you have installed front-ends. The GNU compiler uses the following utilities:

- `cpp` GNU preprocessor that processes all the header files and macros that your target requires (refer to Section 2.5 *`cpp`, the GNU Preprocessor*).
- `as` GNU assembler that produces binary code from assembly language code and puts it in an object file (refer to Section 2.6 *`as`, the GNU Assembler*).
- `ld` GNU linker that binds the code to addresses, links the startup file and libraries to an object file, and then produces an executable binary image (refer to Section 2.7 *ld, the GNU Linker*).

To invoke the compiler, type:

```
gcc option, option...
```

Each *`option`* signifies input that allows you to control how the program compiles (for example, you can interrupt a compile process at intermediate stages). Use commas to separate options. There are many options available that provide specific types of compiled output, some for preprocessing, others controlling assembly, linking, optimization, debugging, and still others for target-specific functions. For instance, call the compiler with a `-v` option to refer to precisely which options are in use for each compilation pass.

`gcc` implicitly recognizes the following file extensions:

- `.c` (for C source code that must be preprocessed)
- `.C` (for C++ source code that must be preprocessed)
- `.s` (for assembler code)
- `.S` (for assembler code that must be preprocessed).

When referring to C++ compilation, `g++` is its customary name. Because there is only one compiler, it is also accurate to use the `gcc` call, no matter what the language context. The `g++` distinction is more useful when the emphasis is on compiling C++ programs, with the GNU compiler acting not merely as a preprocessor, but building object code directly from your C++ program source.

There is no intermediate C version of the program; avoiding an intermediate C representation of the program means that you get better object code and better debugging information. The GNU debugger works with this information in the object code to give you comprehensive C++ source-level editing capabilities.

When you compile C or C++ programs, the compiler inserts a call at the beginning of `main` to a `_ _main` support subroutine. To avoid linking to the standard libraries, specify the `-nostdlib` option. (Including `-lgcc` at the end of your compiler command line resolves this reference and links only with the compiler support library `libgcc.a`; ending your command's input with it ensures that you get a chance to link first with any of your own special libraries).

`_ _main` is the initialization routine for C++ constructors. Because GNU C is meant to interoperate with GNU C++, even C programs must have this call; otherwise, C++ object files linked with a C `main` might fail.

Compilation can involve up to four stages, always in the following order:

- preprocessing
- compiling
- assembling
- linking

The first three stages apply to an individual source file: preprocessing establishes the type of source code to process, compiling produces an object file, assembling establishes the syntax that the compiler expects for symbols, constants, expressions and the general directives. The last stage, linking, completes the compilation process, combining all object files (newly compiled, and those specified as input) into an executable file.

For more information on the GNU compiler and its options, refer to *Using the GNU Compiler Collection (GCC)*.

2.5. cpp, the GNU Preprocessor

cpp is a C-compatible macro preprocessor that works with the GNU compiler to direct the parsing of C preprocessor directives. Preprocessing directives are the lines in your program that start with a # directive name (a # sign followed by an identifier). For instance, cpp merges #include files, expands macro definitions, and processes #ifdef sections. Another example is #define, a directive that defines a macro (#define must be followed by a macro name and the macro's intended expansion).

To refer to the output of cpp, invoke gcc with the -E option; the preprocessed file will print on stdout

The C preprocessor provides the following separate facilities:

- *Inclusion of header files* — Declarations that can be substituted into your program.
- *Macro expansion* When defining macros, which are abbreviations for arbitrary fragments of C code, the C preprocessor replaces them with their definitions throughout a program.
- *Conditional compilation* — Using special preprocessing directives, include or exclude parts of a program according to replaceableious conditions.
- *Line control* — Using a program to combine or rearrange source files into an intermediate file that is then compiled, use line control to provide a source line's origin.

There are two convenient options to assemble files that require C-style preprocessing. Both options depend on using the compiler driver program, gcc, instead of directly calling the assembler.

- Name the source file using the extension .S (capitalized), rather than .s (the .S indicates an assembly language program that requires C-style preprocessing).
- Specify a source language explicitly for a situation, using the -xassembler-with-cpp option.

For more information on cpp, refer to *Using cpp, the C Preprocessor*.

2.6. as, the GNU Assembler

as, the GNU assembler, translates text-based assembly language source files into binary-based object files. Normally it operates without you being aware of it, as the compiler driver program (gcc) invokes it automatically. However, if you are creating your own assembler source files, you must invoke as directly.

If, while using gcc, you want to pass special command-line options to the assembler to control its behavior, you need to use the '-Wa,<text>' command-line option. This option passes <text> directly to the assembler's command line. For example:

```
gcc -c -g -O -Wa,-alh,-L file.c
```

passes the -alh command line option on to the assembler. (This causes the assembler to emit a listing file that shows the assembler source generated by the compiler for each line of the C source file *file*.c).

For more information, refer to *Using as, the Gnu Assembler*.

2.6.1. Object Files

The assembler creates object files that, by convention, have the `.o` extension. These are binary files that contain the assembled source code, information to help the linker integrate the object file into an executable program, debugging information and tables of all of the symbols used in the source code.

Special programs exist to manipulate object files. For example, `objdump` can disassemble an object file back into assembler source code and `ar` can group together multiple object files into an archive or library file.

2.6.2. Assembler Directives

Assembler directives are commands inside the assembler source files that control how the object file is generated. They are also known as pseudo-ops, or pseudo-operations, because they can look like commands in the assembler programming language.

Assembler directives always start with a period (.). The rest of their name is letters, usually in lower case. They have a wide range of different uses, such as specifying alignments, inserting constants into the output, and selecting in which sections of the output file the assembled machine instructions are placed.

2.7. ld, the GNU Linker

The GNU linker, `ld`, combines multiple object files together and creates an executable program from them. It does this by resolving references between the different object files, grouping together similar sections in the object files into one place, arranging for these sections to be loaded at the correct addresses in memory, and generating the necessary header information at the start of a file that allows it to be run.

The linker moves blocks of bytes of your program to their load-time addresses. These blocks slide to their addresses as rigid units; their length does not change and neither does the order of the bytes within them. Such a rigid unit is called a section. Assigning runtime addresses to sections is called relocation. It includes the task of adjusting mentions of object-file addresses so they refer to the proper runtime addresses.

Each section in an object file has a name and a size. Most sections also have an associated block of data, known as the section contents. A section may be marked as allocatable, meaning that space should be reserved for it in memory when the executable starts running. A section may also be marked as loadable, meaning that its contents should be loaded into memory when the executable starts. A section which is allocatable but not loadable will have a zero-filled area of memory created for it.

A section, which is neither loadable nor allocatable, typically contains some sort of debugging information. Every loadable or allocatable output section has two addresses associated with it. The first is the virtual memory address (VMA), the address the section will have when the executable is running. The second is the load memory address (LMA), which is the address in memory where the section will loaded. In most cases the two addresses will be the same. An example of when they might be different is when a data section is loaded into ROM, and then copied into RAM when the program starts. This technique is often used to initialize global variables in a ROM-based system. In this case, the ROM address would be the LMA, and the RAM address would be the VMA. To review the sections in an object file, use the `objdump` binary utility with the `-h` option.

Every object file also has a list of symbols, known as the symbol table. A symbol may be defined or undefined. Each symbol has a name, and each defined symbol has an address. If you compile a C or C++ program into an object file, you get a defined symbol for every defined function and global or static variable. Every undefined function or global variable, which is referenced in the input file, will become an undefined symbol. You can refer to the symbols in an object file by using the nm binary utility, or by using the objdump binary utility with the `-t` option.

The linker is controlled by a linker script which is a text file containing commands in a simple language. The main purpose of the script is to describe how sections in the input files should be mapped into sections in the output file and to control memory layout of the output file. When necessary, the linker script also directs the linker to perform other operations.

The linker uses a default script that compiles into the linker executable, if you do not supply one via the `-T` command line option. Use the `--verbose` option to display the default linker script. Certain options (such as `-r` or `-N`) affect the default linker script. Linker scripts are written in a superset of AT&T's Link Editor Command Language syntax.

For more information, refer to *Using ld, the Gnu Linker*.

2.8. gcov, the Test Coverage Tool

`gcov`, the code coverage/basic block profile display tool, enables you to analyze the basic block profile of your program by recording how often each basic block is executed. This information enables you to determine the sections of code on critical paths and the code blocks that are not executed at all.

2.9. gdb, the Debugging Tool

GDB, the GNU debugger, has the principal purpose of allowing you to stop your program before it terminates. If your program terminates, the debugger helps you determine where it failed. To debug a file named *`file.c`*, first compile the file using `gcc -g` *`file.c`* as a command, where `-g` produces the debugging information. You then run the debugger. Refer to Section 3.4 *Debug the Executable*, for the steps involved in debugging.

Set breakpoints with the `breakpoint` command.

Navigate through the program with the `step` command or the `next` command.

The debugger debugs threads, signals, trace information, and other data in a program. Each time your program performs a function call, a block of data (the stack frame), which shows the location of the call, the arguments, and the local variables of the function is generated. The debugger allows you to examine the stack frame to get your program to work.

For more information, refer to *Debugging with GDB*.

2.10. make, the GNU Recompiling Tool

`make`, the GNU recompiling tool, determines automatically which pieces of a large program you need to recompile and then issues commands to recompile them. `make` conforms to *IEEE Standard 1003.2-1992 (POSIX.2)* and is compatible with any programming language whose compiler can run with command line input from a shell. `make` is not limited only to building programs; it can be used for any task in which some files must update automatically whenever associated files change.

To use `make`, you must write a file (a `makefile`) that describes the relationships among files in your program and provides commands for updating each file. In a program, typically, the executable file is updated from object files, which are in turn made by compiling source files.

When using `make` to recompile an executable, the result may change source files in a directory. If you changed a header file, to be safe, you must recompile each source file that includes that header file. Each compilation produces an object file corresponding to the source file. If any source file has been recompiled, all the object files, whether newly made or saved from previous compilations, must be linked together to produce the new executable.

`make` uses the `makefile` database and the last modified files to decide which of the other files needs updating. For each of those files, `make` implements the commands recorded in the data base of the

`makefile`. The `makefile` has rules which explain how and when to remake certain files that are the targets of a particular rule. A simple `makefile` rule has the following form:

```
target... : dependency... command
```

`target` is usually the name of a file that a program generates; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as with the clean command (a command that, deletes all files from a build directory before building). A *`dependency`* is a file that is used as input to create the target. A target often depends on several files. command is activated by make wnen dependancy has changed. A rule may have more than one command, with each command on its own line. Usually a *`command`* is in a rule with dependencies and serves to create a target file if any dependencies change. However, the rule that specifies commands for the target does not require dependencies; for instance, the rule containing the delete command that associates with the target, clean, does not have dependencies. `make` activates commands on the dependencies to create or to update the target. A rule can also explain how and when to activate a command. A `makefile` can contain other text besides rules; a simple `makefile` requires only rules. Rules generally follow the same pattern.

Note that every command line in a `makefile` must begin with a tab character.

2.10.1. Example `Makefile`

```
edit : main.o kbd.o command.o display.o insert.o search.o \
                files.o utils.o
        cc -o edit main.o kbd.o command.o display.o insert.o \
                search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c

kbd.o : kbd.c defs.h command.h
        cc -c kbd.c

command.o : command.c defs.h command.h
        cc -c command.c

display.o : display.c defs.h buffer.h
        cc -c display.c

insert.o : insert.c defs.h buffer.h
        cc -c insert.c

search.o : search.c defs.h buffer.h
        cc -c search.c

files.o : files.c defs.h buffer.h command.h
         cc -c files.c

utils.o : utils.c defs.h
         cc -c utils.c

clean :
        rm edit main.o kbd.o command.o display.o insert.o \
                search.o files.o utils.o
```

The targets in the example `makefile` include the executable file, `edit`, and the `main.o` and `kbd.o` object files. `main.c` and `defs.h` are the dependency files. Each `.o` file is both a target and a dependency. When a target is a file, it needs to be recompiled or relinked if any of its dependencies change. You must first update any dependencies that automatically generate. In the example `Makefile`, `edit`

depends on eight object files; the object file, `main.o`, depends on the source file, `main.c`, and on the `defs.h` header file. A shell command follows each line that contains a target and dependencies, saying how to update the target file; a tab character must come at the beginning of every command line to distinguish command lines from other lines in the `makefile`. `make` does not know anything about how the commands work; it is up to you to supply commands that update the target file properly. All `make` does is execute the commands in the rule you have specified when the target file needs updating.

For more details, refer to *Using `make`*.

2.11. binutils, the GNU Binary Utilities

`binutils`, the GNU binary utilities, include `ar`, `nm`, `objcopy`, `objdump`, `ranlib`, `size`, `strings`, and `strip`. For targets that use the ELF file format, there is also a tool called `readelf`.

There are three binary utilities, `addr2line`, `windres`, and `dlltool`, that are for use with Cygwin, the porting layer application for Win32 development. The most important of the binary utilities are `objcopy` and `objdump`.

- `objcopy` is a tool to convert object and executable files. It can add and remove sections and symbols, but the most commonly used feature is its ability to change the file format. For example, it can convert an ELF or COFF format executable into an S-record or Intel I-Hex format file. These two formats are often used in building ROM images for standalone and embedded systems. For more information, refer to the *objcopy Utility* section in *Using binutils, the Gnu Binary Utilities*.
- `objdump` is a tool to display information about the contents of an object or executable file. It can display symbol tables and section headers and it can also act as a disassembler. `Objdump` also knows about archives and libraries and can be used to display information on all of the object files inside them. For more information, refer to the *objdump Utility* section in *Using binutils, the Gnu Binary Utilities*.

  A few of the more useful options for `objdump` include: `-h` (to display section headers), `-t` (to display symbols), `-p` (to display private header information) and `-d` for assembler (to display a disassembly). `-d` normally only disassembles sections that are expected to contain instructions. To disassemble all sections, use `-D` instead. The option `--prefix-addresses` can be used to print a complete address on each line of the disassembler's output.

2.12. Using info, the Documentation Tools

`info` provides the sources for documentation for the GNU tools; it requires the following tools, including the TeX tools.

Texinfo, **texindex**, **texi2dvi**

Documentation formatting tools. **Texinfo** requires TEX, the free technical documentation formatting tool written by Donald Knuth. Refer to *Texinfo: The GNU Documentation Format* (ISBN: 1-882114 67 1).

`makeinfo`, `info`

Online documentation tools.

`man` pages

Includes documentation on all the tools and programs in this release.

FLEX: A Fast Lexical Analyzer Generator

Generates lexical analyzers suitable for GCC and other compilers. Refer to *Flex: The Lexical Scanner Generator* (ISBN: 1-882114 21 3).

Using and Porting GNU CC

Contains information about requirements for putting GCC on different platforms, or for modifying GCC; includes documentation from *Using the GNU Compiler Collection (GCC)*.

BYacc

Discusses the Berkeley Yacc parser generator. Refer to *Bison Manual: Using the YACC-compatible Parser Generator* (ISBN: 1-882114 44 2).

Texinfo: The GNU Documentation Format

Details TEX and the printing and generating of documentation, as well as how to write manuals in the TEX style.

Configuration program

Describes the configuration program that Red Hat Enterprise Linux Developer Tools uses.

GNU Coding Standards

Elaborates on the coding standards with which the GNU projects develop.

GNU gprof

Details the GNU performance analyzer (only for the Solaris systems).

You have the freedom to copy the documentation using the accompanying copyright statements, which include the necessary permissions. To get the documentation in HTML or printable form, refer to http://www.fsf.org/doc/doc.html and http://www.fsf.org/doc/other-free-books.html.

For more information on using `info` tools (requiring the TEX tools), refer to *Using info*.

2.12.1. Reading info Documentation

Browse through the documentation using either Emacs or the `info` documentation browser program. The information is in nodes, which correspond to the sections of a printed book. Follow them in sequence, as in books, or, using the hyperlinks, find the node that has the information you need. `info` has hot references (if one section refers to another section, `info` takes you directly to that other section, but gives you the ability to return easily. You can also search for particular words or phrases. Use `info` by typing its name at a shell prompt; no options or arguments are necessary. Check that `info` is in your shell path. If you have problems running `info`, contact your system administrator.

To get help with using `info`, type `h` for a programmed instruction sequence, or [Ctrl]-[h] for a short summary of commands. To stop using `info`, type [q].

2.13. Learn More About the Tools

There are many sources to teach you more about GNU tools:

- For general information about the tools, see: http://www.redhat.com/docs/manuals/gnupro/
- For help with using the tools, see Chapter 3 *Developing with Red Hat Enterprise Linux Developer Tools*.

redhat

Chapter 3.

Developing with Red Hat Enterprise Linux Developer Tools

Use the tutorials in this chapter to learn about standard *native* development with the tools.

See http://www.redhat.com/docs/manuals/gnupro/ for details about the tools.

3.1. Create Source Code

To start, create the following sample source code file and save it as `hello.c`. The following sections show you how to compile this file to form an executable and how to run it.

```
#include <stdio.h>

int a, c;

static void
foo (int b)
{
  c = a + b;
  printf ("%d + %d = %d\n", a, b, c);
}

int
main (void)
{
  int b;

  a = 3;
  b = 4;

  printf ("Hello, world!\n");

  foo (b);

  return 0;
}
```

Example 3-1. Source code to save as `hello.c`

3.2. Compile, Assemble, and Link from Source Code

To compile the code, use the following command:

```
gcc -g hello.c -o hello
```

The `-g` option generates debugging information and `-o` specifies the name of the executable to be produced. Both of these can be omitted. Other useful options include `-O` to enable standard optimizations and `-O2` for extensive optimizations. If no `-O` is specified, GCC will not optimize.

See *Using the GNU Compiler Collection (GCC)* for additional basic compiler information.

3.3. Run the Executable

To run the program, use the following example:

```
./hello
```

The program generates

```
hello world!
3 + 4 = 7
```

and returns when the program exits.

3.4. Debug the Executable

To start GDB, use the following commands:

```
gdb hello
```

After the initial copyright and configuration information, GDB returns its own prompt, `(gdb)`. The following is a sample debugging session:

1. To set a breakpoint, type:
   `break main`

   The following output displays:
   `Breakpoint 1 at 0x132: file hello.c, line 15.`

   Note the exact address and line number may vary, depending upon the target architecture being debugged and the exact layout of the C code in the `hello.c` file.
2. To run the program, type:
   `run`

   The following output displays (when the program stops at a breakpoint):
   ```
   Starting program: hello
   Breakpoint 1, main () at hello.c:15
   15 a = 3;
   ```

   Again note that the exact line number and instruction displayed is dependent upon the source code layout.
3. To print the value of variable, `a`, type:
   `print a`

   The following output displays:
   `$1 = 0`
4. To execute the next command, type `next`:
   `next`

   The following output displays:
   ```
   16 b = 4;
   ```
5. To display the value of a again, type:
   `print a`

   The following output displays:
   ```
   $2 = 3
   ```
6. To display the program being debugged, type:
   `list`

The following output displays:

```
12          int
13          main (void)
14          {
15            int b;
16
17            a = 3;
18            b = 4;
19
20            printf ("Hello, world!\n");
21
22            foo (b);
23
24            return 0;
25          }
```

7. To list a specific function code, use the `list` command with the name of the function to be display. For example, type:

   ```
   list foo
   ```

   The following output displays:

   ```
    1           #include <stdio.h>
    2
    3           int a, c;
    4
    5           static void
    6           foo (int b)
    7           {
    8             c = a + b;
    9             printf ("%d + %d = %d\n", a, b, c);
   10           }
   ```

8. To set a breakpoint at line seven, enter the following input (set a breakpoint at any line by entering `break` *`linenumber`*, where *`linenumber`* is the specific line number to break):

   ```
   break 8
   ```

   The following output displays:

   ```
   Breakpoint 2 at 0xf4: file hello.c, line 8.
   ```

9. To resume normal execution of the program until the next breakpoint, type:

   ```
   continue
   ```

   The following output displays:

   ```
   Continuing.
   Hello, world!
   Breakpoint 2, foo (b=4) at hello.c:8
   8 c = a + b;
   ```

10. To step to the next instruction and execute it, type:

    ```
    step
    ```

    The following output displays:

    ```
    9 printf ("%d + %d = %d\n", a, b, c);
    ```

11. To display the value of c, type:

    ```
    print c
    ```

    The following output displays:

    ```
    $3 = 7
    ```

12. To see how you got to where you are, type:

    ```
    backtrace
    ```

    The following output displays:

    ```
    #0 foo (b=4) at hello.c:9
    ```

```
#1 0x15c in main () at hello.c:18
```

13. To exit the program and quit the debugger, type:
    `quit`

3.5. Assembler Listings

The compiler normally turns a text based source file into a binary object file. It is possible however to instruct it to just convert the source code into assembler and stop there. The `-S` option does this. It also possible to instruct the compiler to produce an *assembler listing* as well as an object file. That can be done as follows:

```
gcc -c -O2 -Wa,-al hello.c
```

`-c` tells GCC to compile or assemble source files, but not to link them. `-O2` produces more fully optimized code. These are both optional. `-Wa` tells the compiler to pass the comma-separated list of options which follows it on to the assembler. The `-al` option is an assembler option to request an assembler listing.

This example shows a partial excerpt of an assembler listing for an x386-based target.

```
  29                            .text
  30 0027 90                    .p2align 2,,3
  31                    .globl main
  32                            .type   main,@function
  33                    main:
  34 0028 55                    pushl   %ebp
  35 0029 89E5                  movl    %esp, %ebp
  36 002b 83EC08                subl    $8, %esp
  37 002e 83E4F0                andl    $-16, %esp
  38 0031 83EC0C                subl    $12, %esp
  39 0034 680E0000              pushl   $.LC1
  39      00
  40 0039 C7050000              movl    $3, a
  40      00000300
  40      0000
  41 0043 E8FCFFFF              call    puts
  41      FF
  42 0048 C7042404              movl    $4, (%esp)
```

Example 3-2. Assembly listing excerpt

It also possible to produce an assembler listing that intermixes the original input source code with the assembler instructions produced by the compiler. This can help track down bugs, discover how the compiler handles certain language constructs (such as function calls) and to learn more about assembly language. To do this, just add an *h* to the assembler option like this:

```
gcc -c -g -O2 -Wa,-alh hello.c
```

3.6. Disassembling

The `objdump` tool can be used to to produce an disassembly of an object or executable file. It is used like this:

```
objdump -d hello.o
```

You could also use the `-D` option to produce a dissassembly of all of the sections in an object file, not only those that have been marked as containing instructions, and `-z` to tell `objdump` to disassemble instructions whose value is zero. (Normally such instructions are just skipped).

Index

A

as (GNU assembler)
  overview, 6
assembler directives, 7
assembler listings
  overview, 16
auxiliary development tools
  overview, 4

B

binutils (GNU binary utilities)
  overview, 10

C

compile, assemble, and link tutorial, 13
creating source code tutorial, 13

D

debug the executable tutorial, 14
disassembling
  overview, 16

G

gcc
  overview, 4, 6
gcov
  overview, 8
gdb (GNU debugger)
  overview, 8

I

info (documentation tools)
  overview, 10

L

ld (GNU linker)
  overview, 7
libraries
  overview, 4

M

make
  example makefile, 9
  overview, 8

O

objdump
  disassembles an object file, 16
object files, 7

R

run the executable tutorial, 14

T

tools
  basic principles, 3
  overview, 1, 3
tutorial
  compile, assemble, and link, 13
  create source code, 13
  debug the executable, 14
  overview, 13
  run the executable, 14